Evaporating tasks during execution of dynamically controllable networks

Russell Knight
Jet Propulsion Laboratory,
California Institute of Technology
Pasadena, CA
russell.knight@jpl.nasa.gov

Abstract

We present an extended execution algorithm for executing plans represented using simple temporal networks with uncertainty. We presume that the network to be executed is dynamically controllable. Our extension allows for skipping tasks that can be shrunk to zero duration if subsequent tasks are ready to start execution. The asymptotic time complexity of the technique scales as a polynomial of the number of timepoints that could execute simultaneously.

Introduction

One advantage of formulating plans conservatively is that we can know a-priori that the execution will (or probably will) succeed. One disadvantage of this approach is that we waste time and resources. In practice, we often skip the execution of tasks that lead to a goal if the tasks are deemed to be superfluous. We present a technique for modeling "skippable" tasks and skipping these tasks during execution.

Simple temporal networks (STNs) [2] provide a rich framework to connect inter-related tasks, execute the tasks, and monitor the execution. Unfortunately, STNs presume that the constructors and executors of these task networks have control over the duration of the tasks being executed, which is not always the case. Simple temporal networks with uncertainty (STNUs) [12] increase the representational capability of STNs by including a labeling of those intervals that are uncertain (contingent). Previous work by [7] has shown that we can know that an STNU is executable if our executor only changes those durations that are not uncertain (free) for future tasks, i.e., the STNU is *dynamically controllable*. The execution strategy of [7] presumed that we wouldn't skip tasks—the technique we present here allows us to skip tasks by shrinking their associated durations to 0. This is implemented in the Mission Data System [3] software framework as part of the timepoint firing algorithm.

Importance

If an execution agent can incorporate runtime feedback, this can be used to optimize plan execution in two ways: 1) reduction of resource utilization and 2) reduction of make-span. We can avoid using resources during execution, thus redundancy can be built into a plan but also can be ignored if it is not required. Additionally, we can reduce the make-span of an executed plan with respect to the original plan. Both resource usage and make-span are useful metrics for plan quality, thus our technique provides for on-line plan optimization.

Preliminaries

Simple Temporal Networks with Uncertainty

A simple temporal network (STN) can be represented as a directed, edge-labeled graph $G = (N, E)$ with real-valued, edge-label functions l and u. The nodes of the graph represent *timepoints*. A timepoint refers to a specific, yet possibly unspecified, moment in time. The edges of the graph, along with the l and u labels, represent *temporal constraints* between timepoints. More specifically, the l and u values bound the duration allowed between two timepoints. (Figure 1 shows two timepoints and a temporal constraint pictorially.)

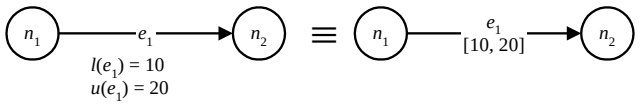

Figure 1 – Timepoint and temporal constraint representation

An *execution* of an STN is a real-valued, node labeling T of G that adheres to the temporal constraints. Thus, for all edges $e = (n_1, n_2) \in E$, $l(e) \leq T(n_2) - T(n_1) \leq u(e)$ implies that T is a valid execution. Checking for STN executability is computable in polynomial time [2]. (This is equivalent to checking consistency of the STN.) But, this assumes that we can know (and control) with certainty the duration of each interval represented by the temporal constraints before execution, which is not the case in many real domains. Thus we need to consider uncertainty for specific intervals.

A simple temporal network with uncertainty (STNU) is an STN with the addition of set $C \subseteq E$. Edges contained in C are temporal intervals that are determined by "nature" (but still adhere to the l and u constraints). We refer to such intervals as *contingent* (as Figure 2), otherwise the

intervals are referred to as being *free* (as Figure 3). Any execution strategy must accommodate the contingent edges. Thus, *T* for an STNU is partially defined by nature, and partially defined by us. There are a number of ways to characterize an STNU with respect to execution. An STNU is *strongly controllable* if we can devise a fixed valued *T* that accommodates all possible assignments by nature to *T*. (We learn nature's assignment after making our assignment.) An STNU is *weakly controllable* if for all possible assignments of *T* by nature an assignment of *T* by us is possible. (We learn nature's assignment before making our assignment.) An STNU is *dynamically controllable* if for all possible assignments of *T* by nature at execution time we can assign (or reassign) *T* for intervals not yet executed. (We learn nature's assignment during execution, and likewise make our assignment during execution.) Strong controllability [12] and dynamic controllability [7] are decidable in polynomial time, while the task of deciding weak controllability is co-NP complete [5].∉∈

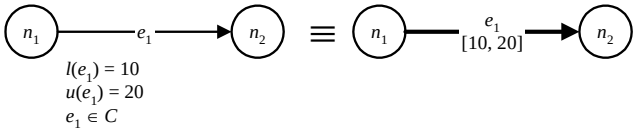

Figure 2 Contingent edge representation

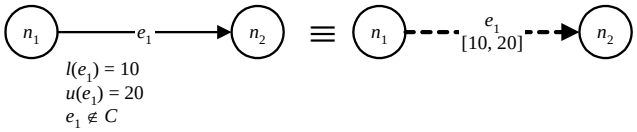

Figure 3 Free edge representation

This paper focuses on the execution of STNUs that are dynamically controllable. With this comes the requirement for *waits*—periods that are determined by nature and the result of computing dynamic controllability. *Waits* cause delays in normally free intervals to ensure the accommodation of the contingent intervals, thus a wait might make a free interval partially contingent. We return to the issue of waits later.

Concerning notation, we use *E* as the edge set of $G = (N, E)$ of the STNU representing the plan being discussed. Also, we presume that for all edges $e \in E$, $l(e) \geq 0$. Any unlabeled edge *e* in our figures is assumed to have $l(e) = 0$ and $u(e) = \infty$.

Plan Representation

A plan is a network of tasks and temporal constraints. A task consists of a temporal constraint $e = (n_1, n_2)$ in the STNU, the timepoints n_1 and n_2 associated with *e*, and the associated values represented by the functions $l(e)$, $u(e)$, and set *C*. We refer to n_1 as the *start-time* of the task, and n_2 as the *end-time* of the task. For simplicity, when referring to a task we will only refer to its associated edge in the STNU. This characterizes the temporal extent of the task but not its preconditions, in-conditions, or effects. To provide these, we include the additional condition-valued, edge-label functions *precond*, *incond*, and *effect*. While conditions have a rich history of semantics and representation, for our purposes a condition is a predicate *holds*(*x*) where *x* is a condition. *holds* considers in the current execution context whether or not a condition holds. If there exists a temporal constraint $e \in E$ that is not related to any task, *holds*(*precond*(*e*)), *holds*(*incond*(*e*)), and *holds*(*effect*(*e*)) are trivially true.

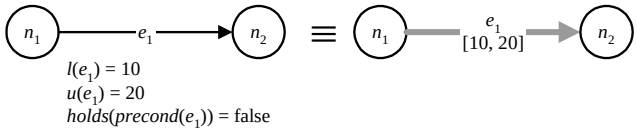

Figure 4 Task with unmet preconditions

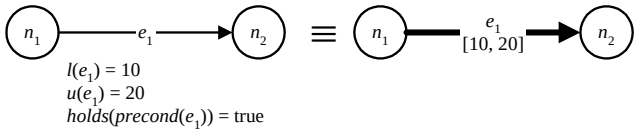

Figure 5 Task with met preconditions

Task Truncatability

To represent that a task is potentially *truncatable*, we employ the use of the edge-predicate function *truncatable*. Note that *truncatable*(*e*) implies that if *e* is being executed, then we can truncate its execution as long as doing so does not force the execution of a task for which its preconditions do not hold. We assume that all free intervals can be truncated. *truncatable*(*e*) being true for some contingent $e \in E$ (where $e \in C$) implies that although the completion time of *e* is under control of nature, its only purpose in the plan is to achieve the preconditions of the next step, and it may be truncated if these preconditions are already met. Thus *truncatable* provides us the semantic information we need to perform opportunistic execution.

Henceforth we will concern ourselves only with *truncatable* edges. Any edge $e \in E$ that is not *truncatable* is contingent, and only nature can determine the interval associated with it. We have no opportunities for skipping it and all possible intervals must be accommodated.

Approach

Execution Strategy

Our overall strategy for execution is to execute timepoints as early as possible while accommodating the contingent intervals. We intend to show how to skip tasks during execution, but first we describe the naïve execution

strategy. The naïve execution for task $e = (n_1, n_2)$ proceeds thusly:

(1) Wait until the start-time (n_1) of the task may be legally assigned to the current time (henceforth referred to as *now*).

(2) Wait until *holds*(*precond*(*e*)) is true.

(3) Assign $T(n_1)$ to *now*, i.e., execute all tasks that n_1 is the starting timepoint.

(4) Assume that *holds*(*incond*(*e*)) continues to be true until *exec*(n_2) is assigned (otherwise an error in execution has occurred and should be handled as an exception).

(5) Wait until the end-time (n_2) of the task may be legally assigned to *now*.

(6) Wait until all edges $x = (n_2, n) \in E$, *holds*(*precond*(*x*)) is true (all tasks for which the end-time of *e* is the start-time have satisfied preconditions).

(7) Assign $T(n_2)$ to *now*.

(8) Assert *effect*(*e*). (Note: effects are not handled during execution as execution is concerned only with preconditions; thus the function *effect* is not pertinent to our discussion and will be dropped.)

(9) Returning to the topic of *waits*, it is clear to see that *waits* can be implemented as part of the *precond* function. Since the system waits for preconditions to hold, each interval associated with the wait must be labeled as being contingent. It should be noted that, when combined, the *precond* and *holds* functions work as a monitor of the state of the world.

Because this framework is built around the dynamic controllability of the STNU, it will execute properly if no free interval is actually contingent and all interval bounds are obeyed by nature. We will assume that this is the case; otherwise some form of plan recovery would be required. Plan recovery is outside the scope of this work.

Simple disjunctive execution

To handle certain types of disjunctions of tasks, we take advantage of an ambiguity in the semantics of time with respect to the preconditions, in-conditions, and effects of tasks. Under normal circumstances, the minimum duration of any task is some real value $\varepsilon > 0$. But, what does it mean semantically when a task is of zero duration? Nothing in the STNU requires tasks to be of greater than zero duration. For our work, we say that a task (or temporal constraint or interval) of zero duration is *skipped*. Thus, if we have a series of possibly zero-duration tasks, all of which are free, then, under certain criteria, we can skip them all. The remainder of this paper describes: 1) under which criteria tasks can be truncated or skipped, and 2) how to tractably execute a plan with skipping.

Skipping Criteria

In general, we assume that we can skip tasks as long as, in the end, the following conditions hold: 1) we introduce no violations of the temporal constraints in the STNU, 2) all newly executing tasks have their preconditions fulfilled, and 3) all currently executing tasks have their in-conditions fulfilled.

We need to identify threats to skipping. Knowing whether or not a temporal constraint violation would occur during execution is handled using the algorithms of [7]. If a task is started with its preconditions fulfilled, it is assumed that its in-conditions will hold during its execution, (otherwise this is an execution-time error that would need to be handled as an exception.) Thus, the remaining threat to skipping is the possibility of a newly executing task not having its preconditions fulfilled.

If a timepoint n_3 comes after or is simultaneous to another timepoint n_1, then the assignment of *exec*(n_3) to *now* implies the assignment of *exec*(n_1) to *now*, thus forcing the execution of n_1. If n_1 has some outgoing edge $e = (n_1, n_2) \in E$ where *holds*(*precond*(*e*)) is false, then executing n_3 causes *e* to fail. Of course, we might be able to skip *e*, but then we would have to check the out-degree of n_2, etc., until we reached the end of our skipping opportunities. Thus, a threat to the execution of n_3 is caused by any edge $e = (n_1, n_2) \in E$ where the execution of n_3 implies the execution of n_1, n_2 cannot be executed due to temporal constraints, and *holds*(*precond*(*e*)) is false. If no such edge exists, then no threats exist, and n_3 can be executed.

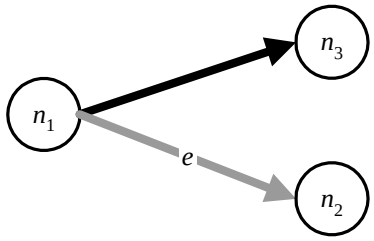

Figure 6 -- Timepoint n_1 threatens n_3

Threat propagation

We can see from the previous discussion that the source of threats is the frontier of the skippable tasks. The frontier of skippable tasks are those constraints or tasks whose start-times could be assigned to now, but whose end-times cannot be due to temporal constraints. Any task that is skippable is not the source of a threat per se—only tasks that are not skippable and for which their preconditions do not hold. It is also important to note that only tasks that are candidates to begin execution are sources of threats. Figure

7 gives an example of the frontier. Task (n_1, n_2) is executing; Tasks (n_2, n_3), (n_2, n_4), (n_3, n_5), (n_4, n_5), and (n_4, n_6) are skippable (they could execute now), and tasks (n_5, n_7) and (n_6, n_7) are at the frontier. (Note: remember that unlabeled edges are ordering constraints.)

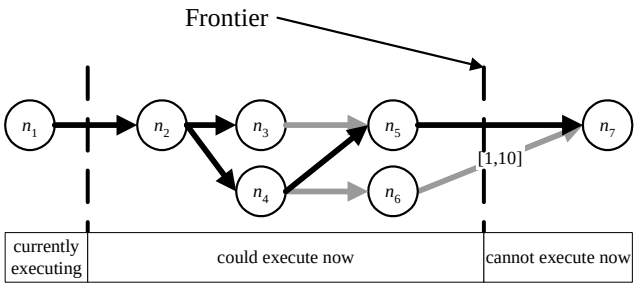

Figure 7 The execution frontier

Theorem 1: The only sources of threats are tasks at the frontier of the collection of skippable tasks.

Proof is by contradiction. Assume the contrary— all tasks at the frontier could execute but there exists a threat to execution in the set of tasks that could execute now. Therefore, there is a threat in the collection of skippable tasks, but all tasks e for which (**a**) the start-time is executable according to the STNU constraints and (**b**) the end-time is not, (**c**) *holds*(*precond*(e)) is true. Then, there must exist an e such that (**d**) its start-time is executable, (**e**) its end-time is executable, (**f**) *holds*(*precond*(e)) is false. We know **d** and **e** because this characterizes the rest of the executable tasks excluded by **a** and **b**. We know **f** because we presume a threat, and by **c** we know it is not at the frontier. But, if the end-time of a task is executable, it can be skipped, and the value of *holds*(*precond*(e)) is no source of a threat, but by our assumption it cannot be skipped. The only reason a task cannot be skipped is if it causes the execution of a start-time of a task e for which *holds*(*precond*(e)) is false. The entirety of the frontier can be executed (by **c**), thus any chain of skippable tasks leads to an executable task, leading us to a contradiction. □

Theorem 2: Threats propagate backward over a task e only when *holds*(*precond*(e)) is false.

Proof: We exhaustively list the alternatives. Consider the tasks $e_1 = (n_1, n_2)$ and $e_2 = (n_2, n_3)$. e_2 is a threat, thus *holds*(*precond*(e_2)) is false, and n_3 is not executable according to the constraints of the STNU.

Case 1: *holds*(*precond*(e_1)) is true and threats propagate backward. This is false because if *holds*(*precond*(e_1)) and n_1 is executable, then we can avoid the threat by executing n_1 and not executing n_2. Thus, threats do not propagate backward over tasks for which the preconditions hold.

Case 2: *holds*(*precond*(e_1)) is false and threats propagate backward. This is true because if we execute n_1 we are in error because *holds*(*precond*(e_1)) is false, and if we skip e_1 by executing n_2 we are in error because *holds*(*precond*(e_2)) is false. □

Figure 8 gives an example of the threat of (n_6, n_7) propagating backward to threaten n_4, but does not threaten n_2.

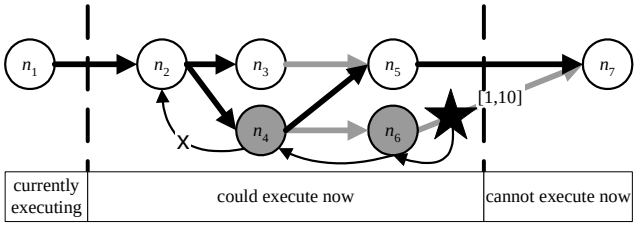

Figure 8 Threats propagate backwards

Theorem 3: Threats propagate forward over all tasks.

Proof is by induction on the number of interceding tasks between a threat and any subsequent executable tasks. Clearly, as our previous example illustrates, the base case of a task $e_1 = (n_1, n_2)$ that begins executing causes the task $e_{\text{threat}} = (n_1, n_{\text{threat}})$ to execute. Since e_{threat} is a threat (*holds*(*precond*(e)) is false and n_{threat} cannot be executed), but shares a start-time with e_1, executing e_1 would lead to the execution of e_{threat}, thus e_1 is also a threat. Inductively, consider an additional $e_m = (n_m, n_{m+1})$. The base case reveals itself at each previous e_x to e_m, leading to each e_x being identified as a threat, until $e_{m-1} = (n_{m-1}, n_m)$ is identified as a threat. The induction closes with e_m being identified as a threat following the same argument as for e_1. It should be noted that since all interceding tasks are skipped except for e_m, the preconditions of the interceding tasks have no effect. The preconditions of e_m also are not important, because even if *holds*(*precond*(e_m)) is true, executing n_m leads to the execution of n_1. □

Figure 9 shows threats propagating forward over (n_4, n_5) even though *holds*(*precond*((n_4, n_5)) is true. Note that propagation would continue to n_3 according to Theorem 2.

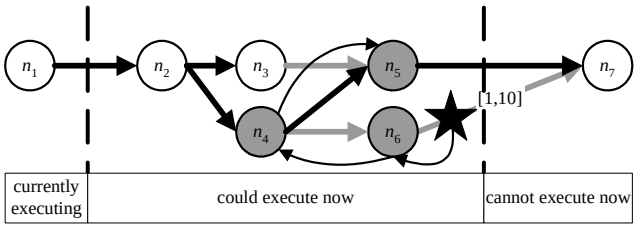

Figure 9 Forward propagation of threats

Threat Propagation Algorithm

Let us call the set of timepoints that can be executed now according to the constraints of the STNU *Candidates* $\subseteq N$. For all currently executing tasks $e = (n_1, n_2)$ such that $e \in C$ and *truncatable*(e) is also true and nature could assign $T(n_2)$ to *now*, we include n_2 in *Candidates*. Let us call the

set of edges that lie on the frontier of the candidates *PossibleThreats*. More formally, *PossibleThreats* $\subseteq E$ such that for each $e = (n_1, n_2) \in$ *PossibleThreats*, $n_1 \in$ *Candidates* and $n_2 \notin$ *Candidates*. Let *Threats* $\subseteq E$ be the set of edges that are known to be threats. We initialize *Threats* with every $e \in$ *PossibleThreats* such that *holds*(*precond*(*e*)) is false (Theorem 1).

We now propagate threats backward across each edge $e \in$ *Candidates* such that *holds*(*precond*(*e*)) is false (Theorem 2), adding each timepoint to *Threats*. We also propagate threats forward across each edge $e \in$ *Candidates* regardless of its preconditions (Theorem 3). This can be accomplished in time that is linear in the number of edges in *Candidates* using a simple reachability algorithm on a transformed graph. After propagation, we have the entire set of threats. We execute all timepoints in *Candidates* that are not start-times for edges in *Threats*.

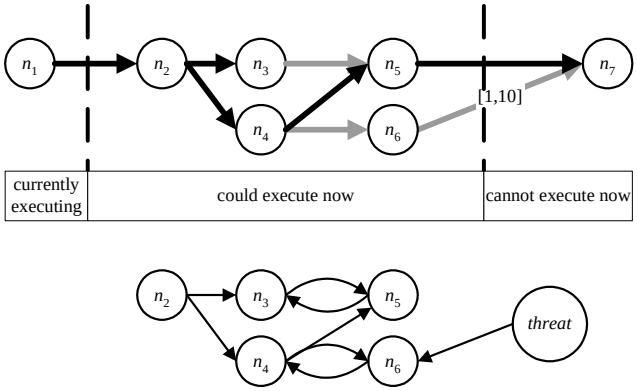

Figure 10 -- Reachability graph transformation

The reachability graph $G_R = (N_R, E_R)$ is a directed graph. *threat* is a node not in *N* from which threats originate. It is from this node that we will compute reachability. N_R=*Candidates* $\cup$ {threat}. For all $e = (n_1, n_2) \in$ *Threats* (as calculated above before propagation), add an edge $x = (n_1,$ *threat*) to E_R. For all $e = (n_1, n_2) \in E$ such that $n_1 \in$ *Candidates* and $n_2 \in$ *Candidates*, add *e* to E_R. If *holds*(*precond*(*e*)) is false, add (n_2, n_1) to E_R. Compute reachability from *threat*, which results in a set of nodes *X* that are reachable from the source node, *threat*. *Threats* is equivalent to *X*. The time complexity of computing reachability is $O(|N|\lg|N|+|E|)$, using Dijkstra's algorithm with a Fibonacci heap [1]. The space complexity is at most $O(|N|)$.

Examples

This system is deployed in a real-world plan execution system. The problems solved by its design are very much a function of the types of problems that needed solving according the to requirements of domain modelers. Some useful examples of meta-structures used by modelers include series preconditional satisfaction and parallel preconditional satisfaction.

Series preconditional satisfaction (SPS) is the idea that several tasks in series are used to achieve a goal, but if a precondition for any task later than the current executing task becomes satisfied, it is appropriate to skip all intervening tasks and execute the latest task that causes no threat. A specific example of this strategy is the "winnowing down" of a state requirement through a series of related tasks. For example, we might want to point a camera on a spacecraft towards a planet. This might require, in general, a series of operations that refine the pointing of the spacecraft. (See Figure 11.) But, if we are already opportunistically pointing at the planet, we might wish to skip steps to continue to our goal, (as in Figure 12).

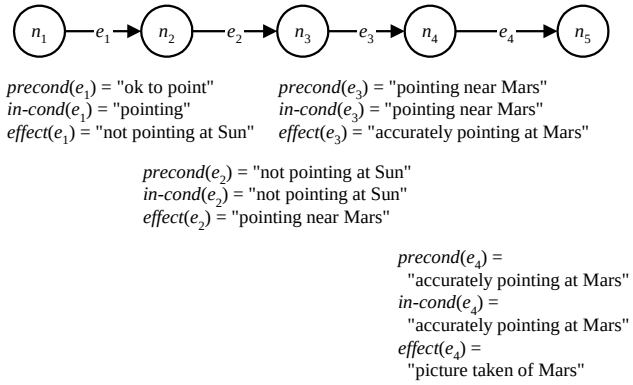

Figure 11 SPS example

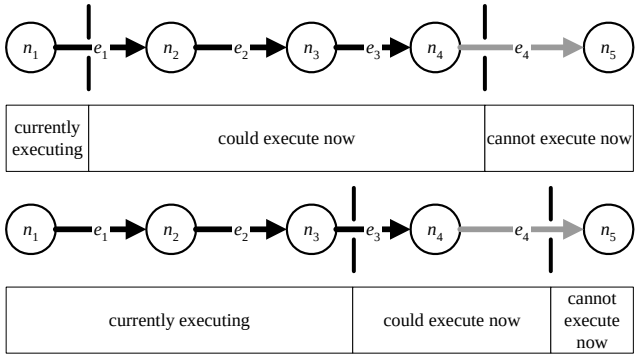

Figure 12 Execution of n_2 and n_3

Additionally, SPS can be augmented to include the idea that at least one of the members of the series must be executed by inserting a temporal constraint from the beginning of the series to the end of the series of $\in$ minimum duration, (as in Figure 13.)

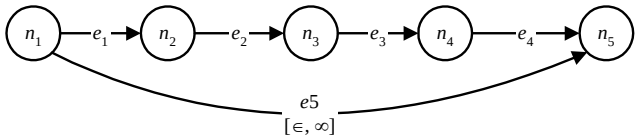

Figure 13 At least one task must execute

Parallel preconditional satisfaction (PPS) is the idea that several tasks in parallel are used to provide redundancy in hopes of satisfying a precondition. Once the precondition is met, the tasks might be truncated or skipped, depending upon whether or not the preconditions of each are satisfied. This allows one task that is ready to start to begin attempting to achieve the goal precondition while other tasks with the same general goal wait for their own preconditions to start. For example, we might want to ensure that the solar panels of a spacecraft are pointed to the sun. We have a number of ways of doing this, and we need only one way to succeed and thus will truncate any other executing tasks and skip pending tasks to continue on with charging the spacecraft. (See Figure 14 and Figure 15.)

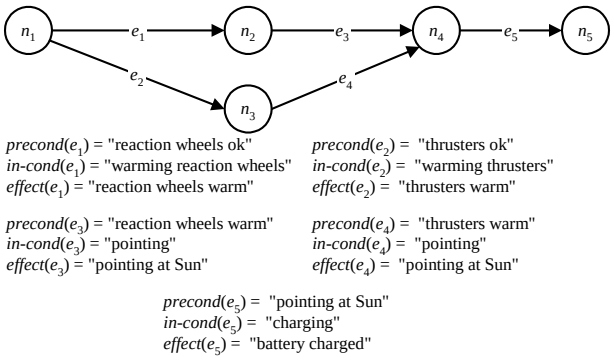

Figure 14 PPS example

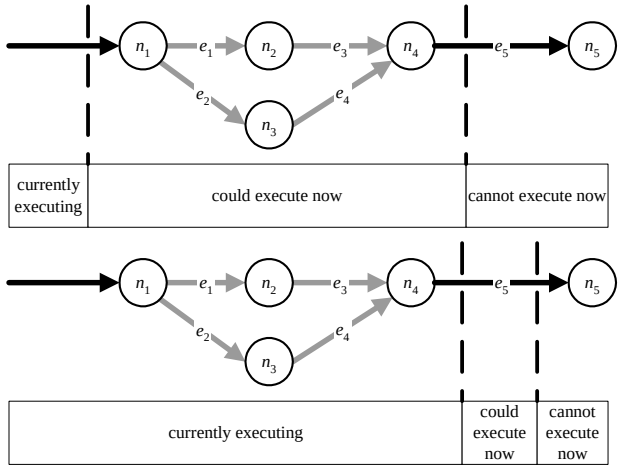

Figure 15 Execution example

Previous Work

The seminal work on STNs is [2]. Previous work in STN and STNU execution includes the work of [6] where efficient algorithms for managing propagation were given and the notion of network dispatchability was introduced. [12] and [5] introduced the notions of various types of controllability for STNUs and proved that determining strong controllability for an STNU is in P while determining weak controllability is co-NP complete.

[7] delivered the surprisingly tractable algorithm for determining dynamic controllability and executing a dynamically controllable STNU. [11] is extending this work to planning with shareable resources by handling some aspects of task sequencing at execution time (on-line).

[9] provides a description of conditional simple temporal networks (CSTNs). Even though we provide a weak form of conditional execution, our plans are not conditional in that all tasks are possible in a single context, meaning that we would have only one context label in a CSTN. Also, our work focuses on the execution of such plans as opposed to the verifiable construction of such plans.

Conclusions

We have described a tractable technique for reasoning about certain disjunctive conditions while executing a plan in metric time. We have presented a plan representation and an algorithm that, when combined with other existing algorithms, provides safe execution. We have provided as examples some useful STNU topologies from real-world examples. Our framework is deployed as part of the Mission Data System [3].

Acknowledgement

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

References

[1] Corman, T., Leiserson, C., and Rivest, R. *Introduction to Algorithms*. MIT Press, 1996, pg. 430

[2] Dechter, R., Meiri I., and Pearl J., "Temporal Constraint Networks," *Artificial Intelligence*, 49, 1991, pp 61-95.

[3] Knight, R., Chien, S., Starbird, T., Gostelow, K., and Keller, R. "Integrating Model-based Artificial Intelligence Planning with Procedureal Elaboration for Onboard Spacecraft Autonomy." *SpaceOps 2000*, Toulouse, France, June 2000.

[4] Laborie, P., Ghallab, M., “Planning with Sharable Resource Constraints,” Proceedings IJCAI-95, 1643-1649

[5] Morris, P., and Muscettola, N., “Managing temporal uncertainty through waypoint controllability.” In T. Dean, editor, *Proceedings of the 16th International Joint Conference on A.I. (IJCAI-99*), pages 1253–1258, Stockholm (Sweden), 1999. Morgan Kaufmann.

[6] Morris, P., Muscettola, N., and Tsamardinos, I., “Reformulating temporal plans for efficient execution.” *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR-98*), Trento (Italy), 1998.

[7] Morris, P., Muscettola, N., and Vidal, T., “Dynamic control of plans with temporal uncertainty.” *International Joint Conference on A.I. (IJCAI-01*), Seattle (WA, USA), 2001.

[8] Muscettola, N., Nayak, P., Pell, B., and Williams, B., “Remote Agent: To Boldly Go Where No AI System Has Gone Before,” Artificial Intelligence 103(1-2):5-48, August 1998.

[9] Tsamardinos, I., Pollack, M., and Horty F., “Merging plans with quantitative temporal constraints, temporally exetended actions, and conditional branches,” *Artificial Intelligence Planning Systems*, 2000.

[10] Vidal, T., “Controllability characterization and checking in contingent temporal constraint networks.” In *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR2000*), Breckenridge (Co, USA), 2000. Morgan Kaufmann, San Francisco, CA.

[11] Vidal, T. and Bidot, J., “Dynamic Sequencing of Tasks in Simple Temporal Networks with Uncertainty.” In the Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP2001), Paphos, Cyprus, 2001. Springer-Verlag.

[12] Vidal, T. and Fargier, H., “Handling contingency in temporal constraint networks: from consistency to controllabilities.” *Journal of Experimental & Theoretical Artificial Intelligenc*e, 11:23–45, 1999.